EventScript: An Event-Processing Language Based on Regular Expressions with Actions

Norman H. Cohen
IBM Thomas J. Watson Research Center
Hawthorne, New York, USA
ncohen@us.ibm.com

Karl Trygve Kalleberg
University of Bergen, Norway
karltk@ii.uib.no

Abstract

EventScript is a simple but powerful language for programming reactive processes. A stream of incoming events is matched against a regular expression. Actions embedded within the regular expression are executed in response to the matching of patterns of events. These actions include assigning computed values to variables and emitting output events. The definition of EventScript presented a number of novel and interesting language-design choices. EventScript has an efficient implementation, and has been used in a development environment for complex event-based applications. We have used EventScript to program both small examples and large industrial applications. Readers of EventScript programs find them easy to understand, and are comfortable with the familiar model of matching regular expressions.

Categories and Subject Descriptors C.3 [**Special Purpose and Application-Based Systems**]: *Real-time and embedded systems;* D.3.4 [**Programming Languages**]: Processors – *Compilers; Run-time environments;* F.4.3 [**Mathematical Logic and Formal Languages**]: Formal Languages – *Classes defined by grammars or automata;* H.2.4 [**Database Management**]: Systems – *Rule-based databases;* I.5.1 [**Pattern Recognition**]: Models – *Structural;* J.7 [**Computers in Other Systems**]: *Command and control; Industrial control; Process control; Real time*

General Terms Algorithms, Performance, Design, Experimentation, Languages, Theory

Keywords event processing; regular expressions; reactive programs; sensors; actuators

1. Introduction

Embedded systems often include reactive processes, which must provide real-time responses to incoming events, such as user-interface events and sensor readings. EventScript is a language for programming reactive processes as regular expressions with embedded actions. For example, the following EventScript program averages readings from sensors S_1, S_2, and S_3. Each time a new reading is received from any one of the sensors, the program emits an event containing the newly computed average:

```
in double S1Input, double S2Input, double S3Input
out double Average

{ v1=0.0; v2=0.0; v3=0.0; }
(
   ( S1Input(v1) | S2Input(v2) | S3Input(v3) )
   { !>Average( (v1+v2+v3)/3.0 ); }
)*
```

The program begins by declaring `S1Input`, `S2Input`, and `S3Input` to be names of input events (corresponding to readings arriving from S_1, S_2, and S_3, respectively) and `Average` to be the name of an output event. Each of these events carries a value of data type `double`. The portion of the program following the event declarations is a regular expression, within which are interleaved actions surrounded by curly braces. The regular expression consists of assignment actions to initialize variables `v1`, `v2`, and `v3`, followed by a regular expression of the form `(...)*`, indicating repeated instances of the regular subexpression inside the parentheses. This regular subexpression consists of three alternatives (separated by the `|` operator), followed by an emit action (denoted by the symbol `!>`). Each alternative matches a different kind of input event and saves the value carried by the event in the corresponding variable. The emit action emits an `Average` event carrying the value of the average of `v1`, `v2`, and `v3`. Besides matching event objects that arrive on an input event channel, EventScript can match events corresponding to the arrival of a specified time.

1.1 Previous approaches

Besides the mature theory and practice associated with regular expressions and finite automata, EventScript builds on two areas of earlier work—the programming of reactive systems and the specification of patterns of events whose occurrences could be recognized as constituting compound events..

1.1.1 Synchronous languages

The bulk of the work in reactive programming has focused on synchronous languages [3]. Synchronous languages address the need to develop programs that can accept unpredictable streams of input from the outside world and react to each input within a bounded amount of time.

The synchronous language Lustre [15] specifies responses to stimuli declaratively, with a set of noncircular equations defining variables in terms of expressions involving other variables. Each variable represents a *flow*—an infinite sequence containing the values associated with that variable on successive clock ticks;

variables declared as input or output variables represent input or output streams, respectively.

The procedural synchronous language Esterel [4] features conceptually concurrent threads that communicate with each other and with the outside world through *signals*. An Esterel execution is a sequence of *instants*, during each of which a signal with a given name is either *present* or *absent*. Among the many Esterel statements are a statement to emit an output signal; a conditional statement testing the presence of a particular signal; a statement to begin execution of a specified body, but to abort that execution in an instant in which a specified input signal is present; a statement to pause until the next instant when a specified input signal is present; and a statement to run two or more statements in parallel. In effect, an Esterel program defines a set of mutually dependent constraints defining the presence of one signal during an instant in terms of the presence of the other signals during that instant. Sophisticated semantic analysis checks that an Esterel program is completely defined, self-consistent, and deadlock-free.

Statecharts [16] are a graphical representation for finite-state machines, featuring a number of powerful notations—such as the clustering of states into "super-states"—that allow machines with many states and transitions to be represented succinctly. Statecharts are intended to specify the behavior of reactive systems. While Statecharts fail, for technical reasons, to be synchronous, a refinement of Statecharts, called Argos [19], is synchronous.

1.1.2 Definition of compound-event patterns

Interest in formalisms to describe patterns of events arose the context of *active databases*, starting with HiPAC [11]. Such databases have *event-condition-action (ECA) rules* specifying that when certain database events such as updates and commits occur, and certain conditions hold, certain actions should be executed. There was a desire to write higher-level ECA rules, by viewing certain combinations of basic events as constituting abstract *compound events*. Interest in compound-event recognition later spread from active databases to the monitoring of business events.

In the COMPOSE event system [13] for the Ode object database, a compound event $E[h]$ is a subset of a *history*, h, of event occurrences. COMPOSE event expressions include a primitive event p (which maps a history h to the set of all occurrences of p in h); $E \wedge F$, where E and F are event expressions (which maps h to $E[h] \cap F[h]$); and $!E$ (which maps h to $h - E[h]$). Other event expressions, whose definitions in terms of event histories are less obvious, include *prior*(E,F) (which takes place when F takes place and E has taken place some time earlier); *sequence*(E,F) (which takes place when E takes place on one event occurrence and F takes place on the next); $(<n>E)$ (which takes place the n^{th} time that E takes place); (*every* $<n>$ E) (which takes place every n^{th} time that E takes place); and $E \mid F$ (which maps a history h to $F[E[h]]$). Named composite events are defined by rules, for example $A(x,y) = \mathit{prior}(B(x),C(y))$, that associate a name with an event expression and specify how attributes of the composite event are computed.

In SAMOS [12], compound events are constructed from other events using sequence, conjunction, disjunction, negation (testing the absence of an event during a specified monitoring interval), and reduction (collapsing occurrences of a specified type of event during a specified monitoring interval into a single occurrence). A *monitoring interval* is defined in terms of starting and ending events or times, or by the union, intersection, or repetition of other monitoring intervals. There are fixed rules defining the attributes of a compound event in terms of the attributes of its constituents. A *coupling mode* determines whether the condition of an ECA rule and any resulting action are processed immediately upon event detection, deferred until the end of the triggering transaction, or executed asynchronously. Programmed priorities determine the order in which multiple eligible rules are executed.

In Amit [1], the rule defining an event pattern is active during a *lifespan* delimited by events or times. Some rules are triggered by the presence or absence of certain sets of event occurrences during the lifespan (e.g., occurrences of each of a specified list of event types, optionally in a specified order; any single occurrence of any of a specified list of event types; the occurrence of at least a specified number of such events, optionally with the stipulation that each event be of a different type; the occurrence of at most a specified number of events of one of a specified set of event types, optionally counting at most one event of each type; or, for two specified event types, an occurrence of the first type and no occurrence of the second type). Other rules are triggered by the passage of time. Each event type in a rule may be accompanied by a predicate that an arriving event must satisfy; a predicate indicating whether the arriving event should contribute to multiple situation occurrences or just one; and a *quantifier* indicating whether to use all eligible events of that type that arrive during the lifespan, only the first, or only the last. There is an option to control whether an event that cannot contribute to a situation as soon as it arrives should be discarded or retained for possible use later. For situations that can be detected before the end of the lifespan, there is an option to timestamp the situation with either the time of detection or the time of lifespan termination. Amit also provides an elaborate set of options for managing lifespan initiation and termination.

In the Rapide event pattern language [18], an input adapter orders events in a partial order called *causal order*, in a manner not specified by Rapide. A *basic event pattern* matches a set consisting of a single event occurrence, with suitable attribute values. More complex patterns consist of two patterns joined by a binary *relational operator*. Three of the relational operators, called *structural operators*, match all unions $P \cup Q$ of event sets such that for all $p \in P$ and $q \in Q$, p and q have a particular temporal or causal relationship. Other relational operators are defined in terms of traditional set operations. A *repetition pattern* specifies a number of repetitions (possibly unbounded), a structural operator, and a subpattern to be repeated, with the stipulation that the structural operator hold between the event sets matched on subsequent iterations. Any pattern may have a predicate that must be true in order for the pattern to match.

Other, similarly intricate mechanisms for specifying compound events include the Snoop event-specification language for the Sentinel database [7], NAOS [10], and Trigs [17].

1.2 Our contribution

Existing approaches to specifying reactive computations and specifying patterns of events use notations and formalisms outside of the programming mainstream. Some researchers in the area have taken pains to prove that their notations are equivalent to regular expressions, but none have explored whether regular expressions themselves might be an appropriate tool for the task at hand. Our experience with EventScript shows that regular expressions are a convenient notation for a wide variety of event-processing tasks.

The programming community is familiar with the use of regular expressions to specify patterns of characters in text. This paradigm can be found in the awk, C#, Java, JavaScript Perl, PHP, Python, Ruby, and Visual Basic languages, for example, as well as in tools like grep, sed, and vi. Therefore, we expect that most programmers will be comfortable with the idea of using regular expressions to specify patterns of event occurrences in event streams. In an age when application development requires mastery of half a dozen or more languages and programming models, developers are grateful to be able to apply a familiar paradigm like regular expressions to event processing, rather than having to learn yet another formalism.

Besides being unfamiliar to mainstream programmers, many of the previous approaches to defining event patterns and programming reactive systems are gratuitously intricate. Rather than consisting of a few well understood basic building blocks, they contain large, *ad hoc*, sometimes redundant collections of specialized features that are hard to understand, and whose applicability to a given programming task may be hard to discover. Our experience with EventScript shows that a far simpler set of primitives suffices to solve real-world event-processing problems.

1.3 The structure of this paper

The remainder of this paper is structured as follows: Section 2 describes the features of EventScript. Section 3 discusses aspects of EventScript programming methodology. Section 4 analyzes language-design dilemmas that arise in a language using regular expressions to specify event processing. Section 5 discusses the implementation and use of EventScript. Section 6 presents the results of performance tests. Section 7 addresses the problem of state-space explosion. Section 8 states our conclusions.

2. Overview of EventScript

An EventScript program receives a sequence of input events and emits a sequence of output events. An event has an event name and carries a value belonging to some EventScript data type. Events of the same name carry values of the same data type. An EventScript run-time implementation establishes a correspondence between these events and entities or occurrences in the outside world, but the language itself does not assume anything about this correspondence. The absence of such assumptions makes EventScript applicable in a wide variety of milieus. For example, the event name might identify the external source or destination of the event, or a port through which the event enters or leaves the system; alternatively, the EventScript run-time implementation might consume and produce external events that already have names (or "event types") associated with them.

An EventScript run-time implementation includes pluggable components called *adapters*. An input adapter is responsible for collecting event signals from the outside world, translating them into EventScript event objects, and feeding them to the EventScript engine. An output adapter is responsible for taking the event objects emitted by the EventScript engine, translating them into some external format, and dispatching them to the outside world.

EventScript has primitive data types, array data types, and structure data types. There are six primitive data types: `boolean`, `double`, `long`, `string`, `time`, and `object` (an opaque object reference that can be taken from an input event or included in an output event value, but whose content cannot be accessed from the EventScript language). The array type t`[]` consists of zero-based arrays with elements of type t. The structure type `{` t_1 n_1`;` ... t_k n_k`; }` consists of structures with fields named n_1, ..., n_k, of types t_1,...,t_k, respectively. (Events that serve as pure signals, carrying no information other than the fact that they occurred, carry values of the empty structure type, `{ }`.)

An EventScript program consists of declarations followed by a single regular expression. Regular expressions are built out of the following parts, where R and R_1, R_2, ..., R_n are regular expressions and i and j are numeric constants such that $0 \le i \le j$:

regular expression	**event sequence matched**
event marker	a single named input event, or the arrival of a particular time (see Section 2.1)
R`*`	zero or more consecutive subsequences, each matching R
R`+`	one or more consecutive subsequences, each matching R
R`?`	either the empty sequence or any sequence matching R
R_1 | ... | R_n	any sequence matching at least one of R_1, ..., R_n
R_1 ... R_n	n consecutive subsequences matching R_1, ..., R_n in that order
R_1 `&` R_2	any sequence matching both R_1 and R_2
R_1 `-` R_2	any sequence matching R_1 but not matching R_2
R`{`i`}`	i consecutive subsequences, each matching R
R`{`i`,`j`}`	from i to j consecutive subsequences, each matching R
R`{`i`,}`	i or more consecutive subsequences, each matching R

An *action block*—a sequence of actions enclosed in `{ ... }` braces—may be interleaved with elements of a sequence regular expression R_1 ... R_n. The action block is executed as it is encountered during the matching of the sequence. There are two kinds of basic actions: The *emit action* `!>Average((v1+v2+v3)/3.0);` emits an output event with the name `Average`, carrying the value of the expression `(v1+v2+v3)/3.0`. An *assignment action* computes the value of a Java-like expression and assigns it to either a simple variable, an array element, or a structure field. In addition, there are conditional actions, repeated actions, and nested action blocks. In the spirit of scripting languages, EventScript variables are not declared. However, each variable must be used in a type-consistent manner.

During the execution of an EventScript program, errors may arise from the arrival of an unexpected event or a problem executing an action or evaluating an expression. In the case of an unexpected event, the offending event is discarded. In the case of a problem evaluating an expression, a default value (depending on the type of the expression) is used. In either case, the program then emits an output event, with the reserved name "`$error`," that describes the problem, after which execution proceeds as normal. The output adapter determines how error events are handled; it may, for example, ignore the error event, display its message on an operator's console, log it, or transmit it to some downstream event consumer as if it were an ordinary event.

Declarations consist of event, data-type, and external-function declarations. Event declarations specify names of input and output events and the types of the data they carry. The declarations

```
in double SetAlarmThreshold,
   { string zoneName;
     double temperature;} ZoneReport
out long PostSpeedLimit, {} CloseGate, {} OpenGate
```

declare `SetAlarmThreshold` and `ZoneReport` to be names of input events, and `PostSpeedLimit`, `CloseGate`, and `OpenGate` to be names of output events. Data-type declarations define names for data types, to facilitate abstraction. The declaration

```
type Point = {double x; double y;}
```

allows the identifier `Point` to be used afterward as a synonym for the structure type `{double x; double y;}`. The identity of a data type is determined by the structure it denotes after expansion of type names. Type definitions can refer to type names declared earlier, but not later, so it is impossible to define recursive types. Thus event objects can be converted to wire representations in a straightforward manner, without performing arbitrary object-graph serialization. Function declarations declare functions that are implemented outside of EventScript (for example, in Java).

2.1 Event markers

The *event marker* `S1Input(v1)` in the example of Section 1 is a placeholder for an occurrence of an event named `S1Input`. The event name in an event marker is followed by parentheses optionally containing a variable into which the value carried by the matched event should be stored. There is also a wildcard event marker, `.` (dot), that matches an event with any name.

In addition to the input events declared by event declarations, there are unnamed input events triggered by the passage of time, carrying values of type `time` that indicate when they occurred. There are two kinds of event markers for time-triggered events: The event marker `elapse[x minutes](t)` is matched when `x` minutes have passed since the previous event. (The time unit may be days, hours, minutes, seconds, milliseconds, or microseconds.) The event marker `arrive[2038-01-19 03:14:08]()` is matched at a specific time and date. The date can be omitted, in which case the marker is matched at the specified time every day. An `arrive` event marker can also contain wildcards (denoted by a dot) or parenthesized expressions in place of components of the date and time. Alternatively, the entire date and time can be replaced by a parenthesized expression of type `time`.

2.2 Event classification

Events arriving with a particular name can be *classified*—that is, assigned new event names based on the values they carry. The following program receives reports of temperature in a refrigeration system and emits events indicating that the system has switched to either a cold state (below 40°) or a warm state (40° or higher). To avoid unwanted storms of output events when the temperature is fluctuating near the 40° boundary, the program emits an output event only when the temperature has crossed 5° past the boundary between the new and old states.

```
in double Temperature
    case {   Temperature < 35.0 ? DefinitelyCold
           : Temperature < 40.0 ? SlightlyCold
           : Temperature < 45.0 ? SlightlyWarm
           : DefinitelyWarm
    }
out {} Warm, {} Cold
{ !>Warm({}); } // initial report
( (.-DefinitelyCold())*
  DefinitelyCold() { !>Cold({}); }
  (.-DefinitelyWarm())*
  DefinitelyWarm() { !>Warm({}); }
)*
```

The input-event declaration for `Temperature` contains an *event-case clause* specifying that input events named `Temperature` should be renamed with one of the event names `Definitely-Cold`, `SlightlyCold`, `SlightlyWarm`, or `DefinitelyWarm`, depending on the value carried by the `Temperature` event. Within a condition of the event-case clause, the original name of the arriving event (in this example, `Temperature`) represents the value carried by the event to be classified. The "else part" of an event-case clause can be omitted, in which case any arriving event that does not satisfy any of the conditions is discarded.

2.3 Event grouping

Event grouping is the splitting of a stream of events into substreams according to the values carried by the events, and the matching of patterns independently within each substream.

Consider a system with sensors that detect motion in any of several square regions. When motion is detected, a `Motion` event is received, giving the region name and (*x*,*y*) coordinates—integers in the range 0 to 99—of the point in the region where motion was detected. Each region is divided into 25 *zones* each measuring 5×5 coordinate units. Whenever motion is detected within the same zone three or more times in one minute, the system is to issue a `ZoneAlert` event specifying the region name and an integer in the range 0 to 24 identifying the zone within the region. The following program groups `Motion` events by zone, so that a different state-machine instance handles each zone of each region:

```
 1|in {string region; long x; long y;} Motion
 2|  group(Motion.region, Motion.x/20, Motion.y/20)
 3|
 4|out {string region; long zone;} ZoneAlert
 5|
 6|{ THRESHOLD=3; }
 7|
 8|( { count = 0; }
 9|  ( Motion() { count=count+1;} )*
10|  arrive[.:.:00]()
11|  { count>=THRESHOLD ?
12|       !>ZoneAlert
13|          ( { region: group[0],
14|              zone: 5*group[1]+group[2] } );
15|  }
16|)*
```

(The conditional action on lines 11–14 emits an action only when the boolean expression on line 11 is true. The expression on lines 13–14 constructs a structure value, given values for each field.)

The *group clause* `group(..., ..., ...)` on line 2 indicates that `Motion` events will be grouped according to the value of a key consisting of three parts: `Motion.region` (the region name), `Motion.x/20` (the row index of the zone within the region, ranging from 0 to 4), and `Motion.y/20` (the column index of the zone within the region, ranging from 0 to 4). (Within the group clause, the identifier `Motion` denotes the value carried by the incoming `Motion` event.) For every resulting key value, there is, in effect, a distinct instance of the regular expression's state machine, with its own copy of the variable `count`. An expression of the form `group[`*n*`]`, where *n* is a zero-based index into the parts of the grouping key, gives the value of the *n*th grouping-key part for the current state-machine instance, as on lines 13 and 14.

Each group clause in a program must have the same number of key components, and corresponding key components must have the same data type in each group clause. If some input-event declarations have grouping clauses and some do not, instances of the events without grouping clauses are *broadcast* to each currently active instance of the state machine.

3. EventScript Programming Methodology

Numerous EventScript programming examples are presented in [9], illustrating idioms for problems such as detecting the absence of an event in a specified interval, debouncing fluctuating input, and smoothing sequences of sensor readings, among others. In this section, we focus on two fundamental aspects of EventScript programming methodology—modeling the structure of the input event stream and decomposing an event-processing problem into multiple stages of a pipeline.

3.1 Modeling the input stream

An EventScript program can be viewed as a model of the stream of events that will arrive during the lifetime of the program, reflecting not only the event sequences for which actions should be performed, but also the event sequences that should be ignored.

Consider, for example, the problem of ensuring that a jet engine is inspected after each use. A `Use` event arrives when the engine is started and an `Inspection` event arrives when completion of an inspection is detected. A `Violation` event is to be emitted whenever two `Use` events are detected without an intervening `Inspection` event. Since the arrival of an input event that does not match a regular expression results in a run-time error, the following attempt to solve this problem is incorrect:

```
in {} Use, {} Inspection
out {} Violation
```

```
( Use() Use() { !>Violation({}); } )*
```

The following regular expression works correctly:

```
 (   Use()
    ( Use() { !>Violation({}); } )*
    Inspection()
  |
    Inspection()
)*
```

3.2 Pipelining EventScript programs

Many event-processing problems can be simplified by decomposing them into stages of a pipeline, in which the output events emitted by one stage are fed as input events into the next stage. Simplification may result, for example, from separating different aspects of the problem into different stages, filtering out irrelevant information, or grouping data in different ways at different stages.

Suppose a device issues status reports every few minutes, reporting either normal or abnormal status. A *window* begins every hour on the hour. When we observe three consecutive windows containing at least one abnormal reading, we issue an alarm and reset the count of consecutive abnormal windows to zero. The alarm should be issued as soon as the first abnormal status report of the third consecutive abnormal window is received.

This problem can be solved by decomposing the processing into two stages. The first stage consumes `NormalStatus` and `AbnormalStatus` events, and emits either a `NormalWindow` event or an `AbnormalWindow` event at the end of each window:

```
in boolean Status
   case { Status ? NormalStatus : AbnormalStatus }
```

```
out { } NormalWindow, { } AbnormalWindow
```

```
( NormalStatus()*
  (   arrive[.:00]() { !>NormalWindow({}); }
    |
      AbnormalStatus() { !>AbnormalWindow({}); }
      .*
      arrive[.:00]() // new window
  )
)*
```

The second stage reads `NormalWindow` and `AbnormalWindow` events and emits `Alarm` events after blocks of three consecutive `AbnormalWindow` events:

```
in { } NormalWindow, { } AbnormalWindow
out { } Alarm
```

```
( ( AbnormalWindow(){0,2} NormalWindow() )*
  AbnormalWindow(){3} { !>Alarm({}); }
)*
```

We have separated concern with determining when a window ends and whether the window is abnormal from concern about whether there have been three consecutive abnormal windows. (Each repetition of the pattern

```
AbnormalWindow(){0,2} NormalWindow()
```

represents a "false start" in which we see a string of up to two consecutive `AbnormalWindow` events that is interrupted by a `NormalWindow` event. After zero or more such false starts, we eventually see three consecutive `AbnormalWindow` events, and emit an `Alarm` event.)

4. Language Design Issues

The specification of event processing through annotated regular expressions presents a unique set of language-design issues. This section discusses interesting issues that arose in the definition of EventScript, and the solutions we chose. As language designers, we considered intricate interactions and pathological cases, but our goal was to arrive at a simple and intuitive language definition that shields the programmer from such considerations.

4.1 Ambiguous regular expressions

An EventScript program may be *ambiguous*. That is, for a given sequence of input events, there may be multiple paths through its regular expression, possibly with different actions along each path. For example, an `A` event may match either alternative in the following program:

```
// Ambiguous Example 1:
```

```
in {} A
out long B
```

```
( A() {x=1;} | A() {x=2;} ) {!>B(x);}
```

The value carried by the emitted `B` event depends on which path is chosen. Ultimately, an ambiguity in a regular expression is attributable to one of two circumstances:

- The next arriving event can be interpreted as matching more than one alternative in a set of alternatives, as in Ambiguous Example 1.
- In a sequence regular expression containing a repetition, the next arriving event can be interpreted as matching either the beginning of the repetition or the beginning of the part of the sequence following the repetition, as in the regular expression `( A() B() )* A() C()`.

We permit ambiguous regular expressions in EventScript, because sometimes an ambiguous regular expression is considerably shorter or clearer than an equivalent unambiguous regular expression (as we shall see in Section 4.1.2), and sometimes (as proven in [5]) there is no equivalent unambiguous regular expression. To ensure that programmers are aware that they have written ambiguous regular expressions, our compiler issues warning messages pointing out the source of the ambiguity. EventScript has rules, described in the following subsection, that define a single, unambiguous behavior for an ambiguous regular expression.

4.1.1 The "all feasible paths" rule

The execution of an EventScript program *follows all feasible paths simultaneously*; on a transition that causes actions to be reached on multiple paths, all actions reached are executed. To make program behavior deterministic, we stipulate that these actions are executed in the order of their textual occurrence in the program. Thus, when Ambiguous Example 1 receives an `A` event, it executes first the action `x=1` and then the action `x=2`, leaving `x` holding the value 2 when the action `!>B(x);` is executed. The all-feasible-paths semantics follows naturally from taking a nondeterministic finite automaton (NFA) for an ambiguous regular expression and applying the classical construction of a deterministic finite automaton (DFA) from the NFA [21], in which there is a one-to-one correspondence between the DFA state reached for a given input and the *set* of NFA states reached for that input.

Actions may be reached along paths followed within regular expressions that ultimately are not matched. The program

```
in {} A, {} B, {} C
out {} Succeed, {} Fail
( A() .* B() - .* C() C() {!>Fail({});} .* )
{!>Succeed({});}
```

matches any sequence of `A`, `B`, and `C` events that starts with an `A` event, ends with a `B` event, and does not contain two consecutive `C` events. (The pattern `.*` matches any sequence of input events.) The program will emit a `Fail` event after any `C` event that is immediately preceded by another `C` event.

Classical regular-expression identities must be applied with caution in the presence of actions executed along all feasible paths. For example, the regular expression

```
A() {!>Z({});} ( B() | C() )
```

emits one `Z` event upon the arrival of an `A` event, but

```
( A() {!>Z({});} B() | A() {!>Z({});} C() )
```

emits two.

4.1.2 The usefulness of ambiguity

Much of the expressive power of regular expressions comes from ambiguity. For example, suppose we have a keypad with keys labeled 0 through 9, we receive a `Key` event carrying the value *x* whenever the key labeled *x* has been pressed, and we wish to emit an `Unlock` event whenever the key sequence 0, 0, 7 is pressed in the middle of an arbitrary sequence of key presses. The following ambiguous program solves this problem in a straightforward way:

```
// Ambiguous Example 2:
in long Key
   case { Key==0 ? K0 : Key==7 ? K7 : Other }
out {} Unlock
( .* K0() K0() K7() { !>Unlock({}); } )*
```

The following unambiguous program has the same behavior, but it is harder to write and to understand:

```
in long Key
   case { Key==0 ? K0 : Key==7 ? K7 : Other }
out {} Unlock
(  ( K7() | Other() )*
   K0()
   ( ( K7() | Other() )+ K0() )*
   K0()+
   ( K7() { !>Unlock({}); } | Other() )
)*
```

An ambiguous regular expression is innocuous as long as the set of feasible paths through the regular expression has collapsed back into a single path at all points where an action is reached. (Ambiguous Example 2 has this property. Ambiguous Example 1 does not.) In some cases, even the execution of actions on different paths in response to a single input event can be innocuous. The behavior of a program becomes much harder to understand if the actions along one path read or modify a variable modified along another, simultaneously feasible, path. The rule that such actions are executed in order of lexical occurrence makes the behavior of such a program deterministic, but we do not encourage programming styles that depend subtly on lexical order.

In Section 3.1, we saw the following incorrect regular expression, in which an `Inspection` event is never expected:

```
( Use() Use() { !>Violation({}); } )*
```

This pattern can be corrected simply by adding `.*` to the front of the pattern and then adding another `.*` as a parallel alternative:

```
( .* Use() Use() { !>Violation({}); } )* | .*
```

This constructive use of ambiguity is reminiscent of *orthogonal products* in Statecharts [16], in which the state of a box consists of the states of several contained boxes responding independently to each input, and of the concurrency operator `||` in Esterel [4].

4.2 Greedy execution of actions

In responsive applications, it is not feasible to defer actions until later input events resolve ambiguities. Therefore, we perform an action as soon as it is reached: The program

```
in {} A, {} B, {} C
out {} Z
( A() B() | A() {!>Z({});} C() )
```

emits a `Z` event as soon as an `A` event arrives, rather than waiting for the next event to determine which alternative should have been selected. Furthermore, since an action can have an irreversible external side effect—the emitting of an output event, possibly to be processed by a system that has performed some physical action in response—we cannot backtrack when the arrival of a later event establishes that one of the paths taken was wrong.

This "greedy" execution of actions arises not only in alternatives, but also in repetitions. After the program

```
in {} A, {} B, {} C
out {} Y, {} Z
A() ( { !>Y({}); } B() )* { !>Z({}); } C()
```

encounters an `A` event, it is poised to iterate the repetition zero or more times. Since the emit actions for `Y` and `Z` are both reachable, both actions are executed greedily at this point (and, similarly, after each subsequent arrival of a `B` event). If the intent is to execute a `Y` action on only an actual execution of the body of the repetition, and to execute a `Z` action only after all repetitions are done, the regular expression should be written as follows instead:

```
A() ( B(){ !>Y({}); } )* C(){ !>Z({}); }
```

As a general principle, actions should be guarded by event markers for the events whose arrivals are meant to trigger the actions.

The regular expression `( A()* { !>Z(); } )*` has a path with an infinite number of action executions, with no intervening input event. (This path corresponds to zero iterations of the inner repetition on each iteration of the outer repetition.) Greedy execution of actions would imply that the EventScript program should immediately enter an infinite loop emitting `Z` events. EventScript prohibits (and our EventScript compiler detects) any program containing a cyclic path that has actions but no event marker.

4.3 Evolution of the language

To elucidate some of the issues presented by the use of regular expressions to specify event processing and to explain the rationale for the current language definition, this section recounts some of the ways EventScript has changed since its initial definition.

EventScript was devised in response to the complexity of compound event models of the kind found in active databases. Our initial goal was to show that the same kind of event patterns could be detected using a small, familiar set of primitive notions. Thus our initial language design was minimal. As we began to write programs to solve real-world problems, it became evident that the original language definition was *too* impoverished, and made some programming tasks inordinately difficult. To achieve a more appropriate balance, we added features to the language grudgingly, only after being convinced that their semantics were well understood and that their presence simplified the programmer's overall task of understanding and using the language.

4.3.1 Data model

In typical event-processing systems, an event is a message containing a set of named attributes. EventScript originally modeled an event as having a named *event type*, which determined the names and types of the event's attributes. Attributes and variables could only hold variables of primitive types. An event marker assigned values of individual attributes to variables, as in

```
RFIDReading(tagID=>tag, readerID=>reader)
```

—where `tagID` and `readerID` are attribute names and `tag` and `reader` are variables.

We added arrays (along with array allocators, subscripted expressions, a `length` built-in function, and a compound action for looping over the elements of an array) to enable such computations as a sliding average over a large or dynamically determined number of sensor readings.

Later, several considerations drove us to add structures (along with a field selection operator and an operator for constructing a structure value from its field values):

- Structures are simple and well understood. Because they are useful for data abstraction (e.g., abstracting the *x* and *y* coordinates of a point into a single entity), their absence was unnatural once we had added arrays.
- We encountered many uses of events to transmit a single primitive value, such as a string or number. Rather than requiring such events to have a single named attribute, it would be simpler to say that an event carries a single value of some data type, and to replace events with multiple attributes by events carrying structure values.
- A structure with no fields is a natural way to represent an event that carries no data, but serves as a pure signal.
- Adding structure types allows us to separate the notion of "event type" into two distinct notions—the name of an event and the data type of the payload it carries, thus providing more flexibility for an environment containing EventScript programs to integrate with the EventScript type system. In particular, the surrounding environment can allow events produced by one EventScript program to be consumed by another EventScript program using a different event name, provided that the payload types match.

By enabling the replacement of event attributes with a unitary event payload of an arbitrary data type, the introduction of structure types actually simplified EventScript. Event markers and emit actions became simpler and more uniform. The size of the compiler and the size of the language definition shrank.

4.3.2 Filtering

Originally, there was no event-case clause in EventScript, but an event marker for a named event could have a predicate whose value for a given event occurrence would determine whether that event occurrence would be processed or ignored. For example, in a state in which the event marker `Reading(degrees=>x) [x>100]` was reachable, the arrival of a `Reading` event would cause the value of its `degrees` attribute to be stored in the variable `x`, after which the predicate `x>100` would be evaluated. If the predicate were true, the arriving event would match the event marker; if there was at least one reachable event marker with the name of the arriving event, but none for which the predicate was true, the arriving event would be ignored; if there were no reachable event markers with the name of the arriving event, the arriving event would be treated as unexpected.

One problem with this approach is that the copying of the `degrees` attribute value to `x` necessarily occurred before the evaluation of the predicate `x>100`. It was overly expensive to back out the copying of attribute values in event markers whose predicates turned out to be false, but, it was unintuitive that the arrival of a supposedly ignored event could still have the side effect of leaving `x` with a new value.

A more serious problem was the possibility of event markers with the same event name having predicates that were not (or could not be determined statically to be) mutually exclusive:

```
( A(attr=>x)[x>10] B() | A(attr=>y)[y<20] C() )
```

Such regular expressions preclude the construction of a DFA whose transitions are labeled by input-event names. Without knowing the value of its `attr` attribute, we cannot determine whether an arriving `A` event should trigger a transition to a state in which only a `B` event, only a `C` event, or either is expected. An alternative would be to use an input alphabet with a distinct symbol for each combination of an event name and a subset of the event-marker predicates appearing with that event name in the program. In effect, there would be four distinct, mutually exclusive, alphabet symbols for A events—

- `A(w)[!(w>10) && !(w<20)]`
- `A(x)[x>10 && !(x<20)]`
- `A(y)[!(y>10) && y<20]`
- `A(z)[z>10 && z<20]`

—transitioning to different states. This approach potentially leads to an exponential blowup in the number of input symbols and a double-exponential blowup in the number of states! The exponential blowup can be avoided if we can ensure that all the predicates appearing in event markers for a given event name are mutually exclusive. The event-case syntax enforces mutual exclusion, because the clause `case` {P_1 ? E_1 `:` ... `:` P_n ? E_n `:` E_{n+1} } associates the event names E_1, ..., E_{n+1} with the mutually exclusive conditions P_1, $\sim P_1 \wedge P_2$, ..., $\sim P_1 \wedge \ldots \wedge \sim P_{n-1} \wedge P_n$, $\sim P_1 \wedge \ldots \wedge \sim P_n$.

4.3.3 Grouping

In the original version of EventScript, the declarations

```
type CallInfo =
  {string areaCode; string number; time callTime;}
```

```
in CallInfo  CallIn
      group(CallIn.areaCode,CallIn.number),
   CallInfo  CallOut
      group(CallOut.areaCode,CallOut.number)
```

would have been written as follows:

```
groupby(AreaCode, Number)
in  CallIn {
      string incomingAreaCode = AreaCode;
      string incomingLocalNumber = Number;
      time incomingCallTime;
   },
   CallOut {
      string outgoingAreaCode = AreaCode;
      string outgoingLocalNumber = Number;
      time outgoingCallTime;
   }
```

Instead of zero or more `group` clauses, each associated with an input-event declaration, an EventScript program had at most one `groupby` clause, containing a list of one or more identifiers, each naming the information in one part of the program's grouping key. In this example, a grouping key consists of a country code, an area code, and a local number. The attribute declaration

```
string incomingAreaCode = AreaCode;
```

constrained the `incomingAreaCode` attribute of a `CallIn` event to hold the value of the `AreaCode` part of the grouping key for the current instance of the state machine.

The replacement of event attributes with unitary event payload values necessitated a different notation. As group clauses are now defined, each part of the grouping key, rather than coming from a particular attribute of an incoming event, can be computed as an arbitrary function of the event payload value. The computation of the n^{th} part of a grouping key is conveyed directly, and can be understood by looking locally within the input-event declaration rather than at a remote list of `groupby` identifiers.

Originally, we required grouping-key computations to be specified for either all or none of the input-event declarations in a program. We later found that certain applications were inordinately difficult to write without broadcast events.

4.4 Bounded-time responses

The salient feature of a synchronous programming language is bounded response time: Given a program in such a language, one can enumerate a finite set of fixed-length instruction paths that may be executed in response to an input. Combined with an instruction-timing model, the finite set of fixed-length paths makes it possible to bound the time required to process any input, and to guarantee that a reactive program meets its real-time constraints.

We undertook the design of EventScript with the intent that it be a synchronous language, but later found a few unbounded-time features to be essential. EventScript fails to be a synchronous language for the following reasons:

- The time required to allocate an array may depend on the length of the array, which may depend on run-time data.
- The number of times a repeated action — for example { `for (i: 0, n-1) a[i]=0;` } — is executed may depend on run-time data (in this case, the value of `n`).
- The time to complete a function call may depend on run-time data. For example, the built-in string-concatenation function has an execution time proportional to the length of its string arguments, and a call on this function within a repetition regular expression could build an arbitrarily long string to be passed to the next call on the function. The execution time of an external function is beyond the control or understanding of EventScript tools.
- The time required to deliver a broadcast event to all currently active state-machine instances depends on the number of currently active instances.

Nonetheless, a large subset of EventScript—including all programs in which array sizes and repeated-action bounds are constant, all function calls are on bounded-time built-in functions, and grouping clauses are present either in all input-event declarations or in none of them—is synchronous. The finite set of fixed-length instruction paths for such programs can be extracted from the DFA transition table generated by the EventScript compiler.

5. Implementation and Use of EventScript

We have developed a Java-based run-time engine that is easily plugged into various execution environments, as well as tools for developing EventScript programs. The tools include a language-sensitive editor, a compiler, and a tester that runs EventScript programs reproducibly in simulated time against a timestamped event trace. We have also defined and implemented a Java API for constructing and decomposing representations of EventScript representations of values and events, used when writing input and output adapters and externally defined functions. The run-time engine, compiler, and other tools are described in detail in [9].

The heart of the run-time engine is a tight loop that uses the current state and the next input event to index into a DFA transition table to find the actions to be executed and the new current state. Thus EventScript is amenable to execution on a resource-constrained platform. Indeed, a summer intern took only a few weeks to write a C program that interpreted the output of the EventScript compiler, so that EventScript programs could be executed on a small mobile device that does not support Java.

EventScript has been incorporated in the Distributed Responsive Infrastructure Virtualization Environment, or DRIVE [8], an environment supporting the construction, testing, distributed deployment, and execution of event-based applications. A DRIVE solution is built out of *components* that consume and emit events through named *ports*. A component may be implemented as a *composite* component, in which case events arriving through input ports of the composite component are fed to input ports of some subcomponents, events emitted from the output ports of some subcomponents are fed to the input ports of other subcomponents, and events emitted from the output ports of some subcomponents are emitted through output ports of the composite component. Alternatively, a component may be implemented as an *atomic* component. The logic of an atomic component may be specified in Java or in EventScript. For an EventScript atomic component, EventScript event names correspond to DRIVE port names.

DRIVE domain-specific component libraries include predefined atomic components for certain real-world producers and consumers of events, such as sensors, actuators, user interfaces, and web services, and DRIVE application developers can write their own Java atomic components for such entities. DRIVE provides a bridge between these real-world entities and EventScript programs. The DRIVE run-time system includes one standard input adapter and one standard output adapter, applicable to all EventScript atomic components; the user of EventScript in DRIVE need not write any EventScript adapters.

EventScript was used extensively in a DRIVE application that uses active RFID tags on forklifts and passive RFID tags on gas canisters, read by readers on the forklifts, to track the movement of canisters within a warehouse, for purposes of industrial safety and supply-chain management. About a dozen EventScript atomic components, each relatively simple on its own, were piped together into an event-processing network performing sophisticated processing. Piping and grouping, used in combination, have proven to be a powerful programming paradigm.

Work has begun to incorporate EventScript in another system, described in [2] and now known as System S, for processing high-volume streams of semistructured data. System S manages the navigation of data through *processing elements* that consume and produce *stream data objects*. Processing elements can currently be written in Java, C++, or a streaming-SQL-like language called SPADE. By wrapping EventScript input and output adapters inside the System S processing-element interface, we will enable the writing of processing elements in EventScript.

6. Performance

We measured four aspects of EventScript run-time performance: throughput, end-to-end latency, scalability with respect to pipeline length, and scalability with respect to the number of distinct grouping key values. All tests ran on a ThinkPad T42p with a Pentium M 2.0 GHz processor and 2.0 GB of RAM, running Windows XP. Tests were invoked from the "Run" menu of Eclipse 3.2.2 and executed on build 2.3 of the IBM J9 implementation of J2RE 1.5.0. The computer was disconnected from the network while the tests were run, and Eclipse was the only active application, but there were several dozen processes running.

6.1 Throughput

We tested throughput on the following four benchmarks:

- **Baseline:** An event object created once and submitted repeatedly to an EventScript program of the form `A()*`.
- **Event creation:** Repeated creation of an EventScript value of type `long` and an event object carrying that value, submitted to an EventScript program of the form `A()*`.
- **Filtering:** Repeated creation of an EventScript value of type `long` in the range 0 to 9 and an event object carrying that value, and submission of that event to an EventScript program that emits output events in response to input events carrying values of 5 or more. We ran three variations, in which none, half, and all of the events carry values that pass the filter.
- **Sliding average:** Repeated creation of an EventScript value of type `double` and an event object carrying that value, and submission of that event to an EventScript program that, for each input event, emits the average of the most recent ten events

We executed 5 "warm-up" runs of each benchmark, followed by 30 runs of one million events each whose results were averaged.

The results of these throughput tests are as follows:

benchmark		events per second
baseline		359,301
event creation		281,187
filter	none pass	281,374
	half pass	253,105
	all pass	235,642
sliding average		95,599

The baseline throughput is competitive with commercial stream-database products. The reduction in throughput due to event creation and filtering is within reasonable bounds. However, it is disconcerting that performance of the sliding-average benchmark—the only benchmark with actions that do "real work"—the throughput falls significantly. In our current implementation, actions and expressions are compiled into Java data structures that are interpreted at run time. In response to the results of the sliding-average benchmark, we have begun work on an alternative implementation in which EventScript actions and expressions are compiled into branches of a Java switch statement that is itself compiled into byte code executed directly at run time.

6.2 End-to-end latency

Our latency tests use a regular expression of the form `( A(n) {!>B(n);} )*`. We tested the end-to-end latency of both a single EventScript program and a pipeline of many EventScript programs. Each run of a latency test consists of one million events. For each event, we obtain the current time, and submit an event carrying that time. The output adapter obtains the time at which an event is emitted, and saves both that time and the received event (containing the event-submission time) in a million-element array. After one million events have been processed in this manner, we postprocess the array to compute the average end-to-end time over all events. We executed 20 such million-event runs. As n increases, the cumulative average per-event latency over the first n runs converges towards 9.700 microseconds per event.

Pipelines of length 2, 3, or occasionally 4 are common in EventScript applications, but for our end-to-end latency tests we used pipelines of up to 50 stages. As expected, end-to-end latency grew linearly (R^2=0.9927) with the length of the pipeline.

6.3 The effect of grouping

The state of an EventScript computation from event to event is saved in a data structure called an *instance context*. We implement grouping by storing instance contexts in hash tables indexed by grouping-key values. When a new event arrives, we compute its grouping key, look up the corresponding instance context in the hash table, and process the event with respect to that context. Our grouping tests examine how this approach scales as the number of groups increases.

The grouping tests execute an EventScript program with the following input-event declaration:

```
in long A group(A)
```

(The grouping key consists of the `long` value carried by the event. Thus there is a separate group for each input-event value.) To test the performance of the program with n groups, we generate one million events, carrying the value i mod n for i ranging from 0 to 999,999. (Thus, when n=1, one million events are submitted to a single group; when n=100, ten thousands events are submitted to each of 100 groups; and when n=1,000,000, one event is submitted to each of one million groups.) We measure the time needed to submit the one million events, using a dummy output adapter.

The time to process an event exhibits very slight logarithmic growth (growing linearly from 3.4 microseconds to 3.9 milliseconds, R^2=0.9389, as the common logarithm of the number of groups increases from 0 to 6). Thus event-processing time is nearly constant with respect to the number of grouping-key values. The slight growth is likely attributable to the cost of rehashing as the hash table reaches its load factor and is expanded.

7. State-Space Explosion

There are classes of regular expressions for which the number of states in the minimal DFA grows exponentially with the length of the regular expression. *We have yet to encounter a real-world problem for which the DFA size is problematic.* Nonetheless, we felt it was important to have a fallback mechanism for any such cases that may arise. That fallback consists of a *hierarchical machine*—a distinct variety of state machine whose structure parallels the structure of the regular expression from which it was generated. The size of a hierarchical machine is proportional to that of the regular expression, but the time for a hierarchical machine to process an input event (not counting the time spent executing actions) is also proportional to the size of the regular expression; for a DFA, this time is independent of the regular expression. By default, our compiler tries to construct a DFA but aborts when the number of states created exceeds 40,000. The user of the compiler is then advised to reinvoke the compiler with options specifying either a higher state limit or the generation of a hierarchical machine.

8. Conclusions

We have applied EventScript to a large body of event-processing problems (see [9]), including examples from in the literature describing other event-processing systems, and we find EventScript to be conducive to elegant solutions. Readers of EventScript programs have remarked that the programs are easy to understand.

On occasion, we have encountered event-processing problems for which it is simple to draw a DFA, but difficult to translate the DFA into a lucid regular expression with suitably placed actions. (In particular, the classical translation algorithms [6, 14, 20, 22] typically yield unintuitive regular expressions, and are not sound when regular expressions are augmented with actions executed on all feasible paths.) The structure of the DFA can always be simplified by moving information implicit in the FSM state into explicit EventScript variables. Taken to the limit, this approach yields a regular expression of the form

(a_1() *action*$_1$ | ... | a_n() *action*$_n$)*

—corresponding to a one-state DFA. A better solution would be an alternative, graphical dialect of EventScript, in which the regular expression is replaced by a (possibly nondeterministic) state-transition graph. The type system, event model, and declarations of EventScript (including event-case and group clauses) would remain, as would the textual syntax of actions, the output of the compiler, and the run-time engine.

Acknowledgments

Han Chen, the principal designer and implementer of DRIVE, did the bulk of the work to integrate EventScript into DRIVE. This work was partially supported by the IT839 project from the Institute of Information Technology Assessment and the Ministry of Information and Communication of the Republic of Korea. Iqbal Mohomed of the University of Toronto wrote the C version of the EventScript engine described in Section 5.

References

[1] Adi, Asaf, Botzer, David, and Etzion, Opher. The situation manager component of Amit—active middleware technology. In Alon Halevy and Avigdor Gal, eds., *Next Generation Information Systems and Technologies: 5th International Workshop, NGITS 2002, Caesarea, Israel, June 24-25 2002, Proceedings. LNCS* **2382**, Springer, Berlin, 2002, 158-168.

[2] Amini, Lisa, Jain, Navendu, Sehgal, Anshul, Silber, Jeremy, and Verscheure, Olivier. Adaptive control of extreme-scale stream processing systems. *26th IEEE International Conference on Distributed Computing Systems (ICDCS '06)*, 2006, 71–77.

[3] Benveniste, Albert, Caspi, Paul, Edwards, Stephen A., Halbwachs, Nicolas, le Guernic, Paul, and de Simone, Robert. The synchronous languages 12 years later. *Proc. IEEE* **91**, No. 1 (Jan. 2003), 64-83.

[4] Berry, Gérard. *The Esterel v5 Language Primer: Version 5.21 release 2.0.* Centre de Mathématiques Appliquées, Sophia-Antipolis, France, Apr. 1999. ftp://ftp-sop.inria.fr/meije/esterel/papers/primer.pdf

[5] Bruggemann-Klein, Anne, and Wood, Derick. One-unambiguous regular languages. *Information and Computation* **142**, 2 (1998), 182–206.

[6] Brzozowski, Janusz A. Derivatives of regular expressions. *J. ACM* **11**, 4 (Oct. 1964), 481–194.

[7] Chakavarthy, Sharma, and Mishra, Deepak. Snoop: an expressive event specification language for active databases. Tech. report UF-CIS-TR-93-007, Dept. of Comp. and Inf. Sci., U. of Florida, Mar. 1993.

[8] Chen, H., Chou, P.B., Cohen, N.H., Duri, S.S., and Jung, C.W. A distributed responsive infrastructure virtualization environment for sensor and actuator applications. *IBM Sys. J.* **47**, No. 2 (May 2008).

[9] Cohen, Norman H. EventScript: Using regular expressions to program event-processing agents. IBM Research Report RC 24387, October 23, 2007

[10] Collet, Christine, and Coupaye, Thierry. Primitive and Composite Events in NAOS. *Actes des 12e Journées Bases de Données Avancées*, Cassis (France), August 1996, 331–349.

[11] Dayal, U., Blaustein, B., Buchmann, A., Chakravarthy, U., Hsu, M., Ledin, R., McCarthy, D., Rosenthal, A., Sarin, S., Carey, M. J., Livny, M., and Jauhari, R. The HiPAC project: combining active databases and timing constraints. *SIGMOD Rec.* **17**, 1 (Mar. 1988), 51-70.

[12] Dittrich, Klaus R., Fritschi, Hans, Gatziu, Stella, Geppert, Andreas, and Vaduva, Anca. SAMOS in hindsight: experiences in building an active object-oriented DBMS. Technical report 2000.05, Database Technology Research Group, University of Zurich Department of Information Technology, ftp://ftp.ifi.unizh.ch/pub/techreports/TR-2000/ifi-2000.05.pdf

[13] Gehani, Narain, Jagadish, H. V., and Shmueli, O. COMPOSE: a system for composite event specification and detection. In Adam, Nabil R., and Bhargava, Barat K., eds., Advanced Database Systems, *LNCS* **759**, 1994, 3–15.

[14] Glushkov, V.M. The abstract theory of automata. *Russian Mathematical Surveys* **16** (1961), 1–53.

[15] Halbwachs, Nicolas, Caspi, Paul, Raymond, Pascal, and Pilaud, Daniel. The synchronous data flow language LUSTRE. *Proc. IEEE* **79**, No. 9 (Sep. 1991), 1305-1320.

[16] Harel, David. Statecharts: a visual formalism for complex systems. *Science of Computer Programming* **8** (1987) 231–274.

[17] Kappel, Gerti, Rausch-Schott, Stefan, and Retschitzegger, Werner. A tour on the TriGS active database system—architecture and implementation. In Proceedings of the 1998 ACM Symposium on Applied Computing (SAC '98), Atlanta, Georgia, Feb. 27 - Mar. 1, 1998, 211-219.

[18] Luckham, David. The Rapide pattern language. In *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, Boston, 2002, chapter 8.

[19] Maraninchi, F. The Argos language: graphical representation of automata and description of reactive systems. *Proceedings, IEEE Workshop on Visual Languages*, Kobe, Japan, Oct. 1991.

[20] McNaughton, R., and Yamada, H. Regular expressions and state graphs for automata. *IRE Transactions on Electronic Computers* **EC-9**, 1 (Mar. 1960), 39–47.

[21] Rabin, M.O., and Scott, D. Finite automata and their decision problems. *IBM Journal of Research and Development* **3**, 2 (April 1959), 114–125.

[22] Thompson, Ken. Regular expression search algorithm. *Commun. ACM* **11**, 6 (June 1968), 419–422.